SANDIA REPORT

SAND2010-1192
Unlimited Release
Printed March 2010

SIERRA Toolkit Computational Mesh Conceptual Model

H. Carter Edwards, Alan B. Williams, Gregory D. Sjaardema, David G. Baur, William K. Cochran

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2010-1192
Unlimited Release
Printed March 2010

SIERRA Toolkit Computational Mesh Conceptual Model

H. Carter Edwards Alan B. Williams Gregory D. Sjaardema
David G. Baur William K. Cochran

Computational Simulation Infrastructure
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185

Abstract

The Sierra Toolkit computational mesh is a software library intended to support massively parallel multi-physics computations on dynamically changing unstructured meshes. This domain of intended use is inherently complex due to distributed memory parallelism, parallel scalability, heterogeneity of physics, heterogeneous discretization of an unstructured mesh, and runtime adaptation of the mesh. Management of this inherent complexity begins with a conceptual analysis and modeling of this domain of intended use; *i.e.*, development of a domain model [3]. The Sierra Toolkit computational mesh software library is designed and implemented based upon this domain model. Software developers using, maintaining, or extending the Sierra Toolkit computational mesh library must be familiar with the concepts / domain model presented in this report.

Contents

Figures

List of Algorithms

Glossary

The following key terms are listed with an abbreviated definition, and the section of this report where the term / concept is fully defined.

Mesh For the domain of intended use, a mesh is an unstructured spatial discretization (Section 1).

Mesh Entity Fundamental unit of a spatial discretization (Sections 1.1 and 2).

Mesh Entity Rank Placement of mesh entities in a composition hierarchy (Section 2.1).

Mesh Entity Identifier Persistent integer identifier for a mesh entity (Section 2.2).

Mesh Entity Relation Directed relationship between two mesh entities in a composition hierarchy (Section 3).

Mesh Entity Closure A set of mesh entities defined by traversing mesh entity relations (Section 3.3).

Mesh Entity Topology A set of expected and well-defined mesh entity relations (Section 3.4).

Mesh Part A subset of mesh entities in the problem domain (Sections 1.1 and 4).

Mesh Part Induced Membership A mesh entity may be induced to be a member of a mesh part through its relations (Section 4.3).

Mesh Part Relation Stencil A mesh entity may be induced to be a member of a mesh part through a stencil of its relations (Section 4.3).

Mesh Field A variable associated with mesh entities (Sections 1.1 and 5).

Mesh Field Type A multidimensional array type defined for one or more mesh fields (Section 5.1).

Mesh Field Restriction A restriction of a field to exist for specified mesh parts (Section 5.3).

Mesh Field Data Data associated with the pairing of a mesh field with a mesh entity (Section 5.2).

Mesh Field Data Relation Stencil A means of quickly accessing field data associated with related mesh entities (Section 5.4).

Mesh Field Data Bucket A contiguous block of similarly defined (*i.e.*, homogeneous) field data associated with a collection of mesh entities (Section 6).

Mesh Meta Data Aggregate A software-architecture motivated grouping of mesh parts, mesh fields, and similar mesh data (Section 8).

Mesh Bulk Data Aggregate A software-architecture motivated grouping of mesh entities, mesh field data, and similar mesh data (Section 9).

Parallel Ownership Mesh entities are distributed among the processes of a parallel application. Every mesh entity is owned by exactly one process (Section 7.1).

Parallel Sharing and Ghosting A mesh entity may reside on more than one process of a parallel application. These mesh entities are either shared or ghosted (Sections 7.1 and 7.2).

Parallel Consistency A mesh entity is parallel consistent if all shared or ghosted copies of that mesh entity are consistent with the owned copy of that mesh entity. A mesh is parallel consistent if all of its mesh entities are parallel consistent (Section 7.3).

1 Fundamental Concepts

The SIERRA Toolkit is a collection of software libraries that provide physics-independent capabilities needed by computational mechanics simulations. The intended domain of computational simulations are those requiring massively parallel computations, multiple coupled models with varied physics and scales, unstructured meshes with multiple types of discretization cells, and solution strategies that dynamically modify the unstructured mesh. A fundamental concept of these simulations is of a *computational mesh* comprised of both the spatial discretization (*e.g.*, the elements, nodes, and connectivity) and field variables discretized over the spatial discretization.

This report presents the concepts or *domain model* [3] for the SIERRA Toolkit Mesh Module. A domain model expresses the requirements for software within a conceptual model of the problem domain. This model spans the software's requirements and architectural design—the decomposition of software into architectural components and relationships among those components.

The concepts synthesized in this domain model are influenced by numerous research and development projects over the past decade. The projects that had the most significant influence include:

- SIERRA Framework [2, 10, 1] at Sandia National Laboratories,
- Unstructured Grid (UG) [5, 6, 16] at the University of Heidelberg, and
- AOMD [9, 11] and its predecessor Trellis [15] at Rensselaer Polytechnic Institute.

1.1 Mesh Entity, Part, and Field

Mesh Entity: A computational mesh contains a collection of *mesh entities* which represent the fundamental, atomic units of the spatial discretization. For example, the mesh entities of a finite element mesh may include nodes, edges, faces, and elements. Mesh entities are connected to one another through directed *mesh entity relations*. For example, an element within a finite element mesh has relations directed from the element to the nodes that define its vertices.

Mesh Part: A problem domain defines subsets of mesh entities, or *mesh parts*, for problem-specific subdomains or other computational subsets of the problem domain. Consider the simple problem of fluid flowing through a pipe with localized heating illustrated in Figure 1. In this illustrative multi-physics problem, fluid computations are applied to the fluid subdomain, deformation computations are applied to the pipe subdomain, and heat transfer computations are applied to the entire domain.

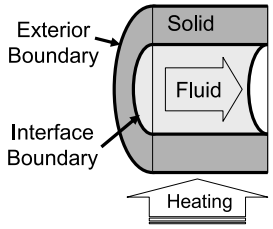


Figure 1. Illustrative multi-physics and multi-part problem of fluid flow through pipe with localized heating.

A spatial discretization is generated from a geometric representation of the problem domain. For interesting problem domains, this representation is typically constructed from a collection of simpler geometric components. These components could be carried through the meshing process to define mesh parts. The mesh parts may not have one-to-one correspondence with a set of computations. Mesh parts will also be defined by the discretization scheme. For example, a traditional simple finite element discretization may partition a three dimensional problem domain into hexahedral mesh entities with node mesh entities at the vertices. This simple discretization scheme induces two mesh parts: a set of hexahedra and a set of nodes.

Mesh Field: *Mesh fields* are the independent, dependent, or scratch variables that are required by the applications' computations, and have data associated with mesh entities. In the illustrative problem (Figure 1, the discretization of a fluid element (Figure 2) could have velocity and pressure $(\underline{\mathrm{V}}, P)$ field data at the vertex nodes, while the mid-edge nodes only have velocity field data $(\underline{\mathrm{V}})$. Such a discretization scheme is referred to as a mixed element formulation.

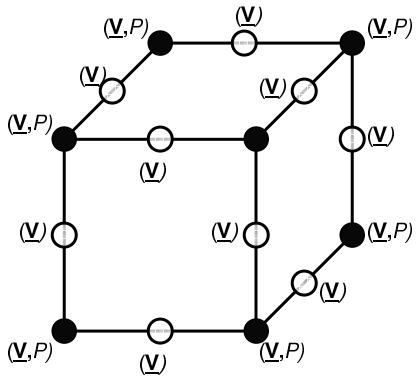


Figure 2. Illustrative fluid element with velocity field data at all node and pressure field data only at vertex nodes

1.2 Domain Model "Big Picture"

A "big picture" diagram of the Sierra Toolkit mesh domain model is given in Figure 3. This diagram identifies components and interrelationships which have not yet been defined or described. As such, this domain model diagram is intended to provide the reader with a "big picture" context to the domain model's components and interrelationships as they are described in the remainder of this document.

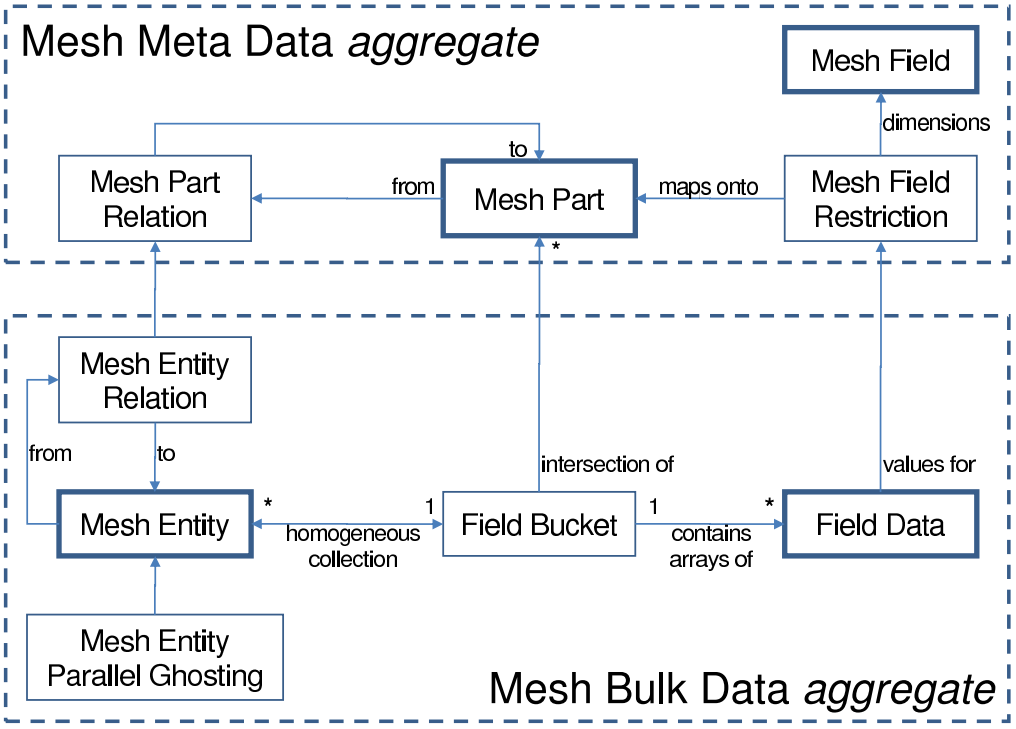


Figure 3. Components and interrelationships of the Sierra Toolkit mesh domain model, as presented throughout this document

1.3 Mesh Meta Data and Mesh Bulk Data Aggregates

The data of a computational mesh are divided into *mesh meta data* and *mesh bulk data*. Mesh meta data defines a specification for the discretization of a problem domain and fields. Mesh bulk data implements a particular discretization conformal to the specification. Given a specification for a computational mesh (mesh meta data) there are numerous admissible implementations (mesh bulk data) which conform to that specification.

Mesh Meta Data: The meta data for a computational mesh discretization includes specifications for: types of mesh entities, mesh parts, mesh fields, and where fields are required

to exist. In Figure 1, mesh meta data specify that the discretization will include elements, nodes, and faces, the existence of interior fluid, enclosing pipe, interface surface, and other mesh parts; the existence of temperature, fluid velocity, fluid pressure, and other mesh fields; and that fluid fields exist only on nodes which are members of the fluid mesh part.

Mesh Bulk Data: A computational mesh's bulk data includes mesh entities and their interrelationships, mesh entities' membership in mesh parts, and mesh field data variables. For the illustrative problem (Figure 1) the mesh bulk data would include elements and nodes exclusively in the fluid part, elements and nodes exclusively in the pipe part, and nodes which are members of both the fluid part and pipe part. The mesh bulk data would also include basis function variables for temperature, fluid velocity, fluid pressure, and other discretized mesh fields. Note that the fluid variables should exist for nodes which are members of the of the fluid part only.

Multiple instances of mesh bulk data are possible for a given mesh meta data specification. For example, a particular discretization of the problem domain could be uniformly refined to generate a different instances which is conformal to the same mesh meta data specification. It is the difference between the *specification* of a computational mesh and the *instances* of a computational mesh which differentiates mesh meta data from mesh bulk data.

Aggregate design principle The architectural decomposition of a computational mesh into meta data and bulk data is derived from the aggregate design principle [3]. This principle is summarized as follows.

> "An *aggregate* is a cluster of associated objects that we treat as a unit for the purpose of data changes. Each *aggregate* has a root and a boundary. The boundary defines what is inside the *aggregate*. The root is a single, specific *entity* contained in the *aggregate*. Choose one *entity* to be the root of each *aggregate*, and control all access to objects inside the boundary through the root. This arrangement makes it practical to enforce all *invariants* for objects in the *aggregate* and for the *aggregate* as a whole in any state change. [Invariants of an aggregate] are consistency rules that must be maintained whenever data changes [and] will involve relationships between members of the *aggregate*." [3]

The aggregate design principle is applied to facilitate enforcement of the complicated invariants (*a.k.a.*, consistency rules) of the mesh meta data and mesh bulk data aggregates. Mapping of this conceptual design into the software design and subsequent implementation supports more robust and maintainable enforcement of invariants. These invariants are described throughout subsequent sections of this report.

1.4 Heterogeneity and Modifiability

A computational mesh can have multi-physics computations, multiple mesh parts, multiple mesh fields which are not uniformly defined over the discretization, and multiple kinds of mesh entities (*e.g.*, hexahedra, wedges, pyramids). Heterogeneities of a computational mesh are captured in the mesh meta data; *i.e.*, through the identified mesh entity types, mesh parts, mesh fields, and rules for laying out fields over the spatial discretization. The complexities and heterogeneities of the application's computations are not part of the computational mesh; however, the computations' spatial discretization and field requirements are realized in the mesh meta data.

An application's solution algorithm can include runtime modifications to the mesh. For example, the mesh may be refined or coarsened to satisfy a discretization error condition or elements on an eroding boundary of a mesh may be removed from the computation. Runtime modifications to a computational mesh include creating or destroying mesh entities, creating or destroying mesh entity relations, and adding or removing mesh entities as members of mesh parts. These changes are limited to modifying the mesh bulk data while conforming to mesh meta data.

1.5 Parallel Computing and Data Distribution

Parallel processing is a critical high performance computing (HPC) capability. The computational environment is assumed to provide at least one of two levels of parallelism: distributed memory processes and/or shared memory threads.

Distributed Memory Processes: The computational environment is assumed to consist of a collection of processing nodes connected through a dedicated high performance communication network. A parallel *process* executes within a particular node and has exclusive ownership over a well-defined memory space within that node. Data are explicitly exchanged between processes' memory spaces through the communication network by the application developer. The *de facto* standard mechanism for explicit data exchange is message passing via a Message Passing Interface (MPI) [14] compliant library.

Shared Memory Threads: A processing node may support multiple parallel *threads* of execution within a process. A thread is a lightweight execution environment within a process that has shared access to some or all of the resources of the process. There are several different multi-core and memory programming models available for threads [7, 13], each affecting software design and implementation differently. The threads of a process could be heterogeneous in that they run on different types of cores (*e.g.* cpu vs. gpu) and access the process memory through different mechanisms. The potential heterogeneity of threads is not addressed in the current scope of this domain model.

Distribution of Mesh Bulk Data: When using distributed memory parallel processing it is typical that the memory required for mesh bulk data will typically be too large to reside in the memory space of a single process. As such the mesh bulk data must be partitioned and distributed among processes, such as with a domain decomposition approach. Domain decomposition exploits the geometric or topological locality of a spatial discretization to group neighboring or connected mesh entities together on a process. Domain decomposition of an unstructured mesh typically results in portions of the mesh, and corresponding parts of mesh entities, being shared among processes as illustrated in Figure 4.

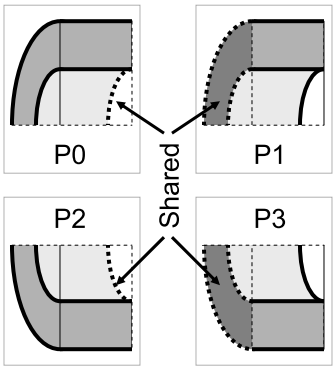


Figure 4. Parallel domain decomposition for illustrative multi-physics problem.

Duplication of Mesh Meta Data: When using distributed memory parallel processing it is typical that the memory required for mesh meta data will typically not be too large to reside in the memory space of a single process. Mesh meta data is duplicated in the memory space of all processes.

2 Mesh Entity

A mesh entity has a role within a computational mesh. In a three dimensional finite element mesh the points, line segments, bounded areas, and bounded volumes of the spatial discretization are represented by node, edge, face, and element mesh entities. Mesh entities are typically composed from, or defined by, other mesh entities. For example, the element mesh entity in Figure 2 can be defined by node mesh entities located at the element's vertices and mid-edges. The composition of mesh entities from other mesh entities defines composition-hierarchy for mesh entities.

2.1 Mesh Entity Rank

A *mesh entity rank* is a non-negative integer which enumerates both the role of a mesh entity in the mesh, and the position of a mesh entity the composition-hierarchy. In a three dimensional finite element mesh the ranks, and corresponding roles, are typically defined to correlate with the topological dimension of the mesh entities; *e.g.*, rank 0 = node, rank 1 = edge, rank 2 = face, and rank 3 = element. A mesh entity's position in a composition-hierarchy is defined by its rank: a mesh entity can only be composed from mesh entities with lesser ranks. For example, an edge (rank 1) can be composed of nodes (rank 0), but an edge cannot be composed of faces (rank 2).

The application-defined mesh entity ranks are part of the mesh meta data. The integer values for mesh entity ranks are in the range $[0, N]$, where the upper bound of the range is specified by the application within the mesh meta data. For example, an application may extend the traditional finite element mesh entity roles with constraint mesh entities = rank 4 and patch mesh entities = rank 5.

2.2 Unique Identification and Ordering

A mesh entity ($entity_a^J$) is uniquely identified within a computational mesh by its integer rank J and integer identifier a, as per Equation 1. For example, a finite element mesh may have a mesh entity identified as node #5 ($entity_5^0$) as well as another mesh entity identified as element #5 ($entity_5^3$), but at most one of each.

$$\begin{aligned} entity_a^J \rightarrow \begin{cases} J & \in & [\,0 \,..\, \text{number of application-defined mesh entity ranks}\,) \\ a & \in & [\,1 \,..\, \text{maximum value defined by implementation}\,] \end{cases} \\ entity_a^J = entity_b^K \iff (J,a) = (K,b)\,. \end{aligned} \tag{1}$$

A computational mesh is not required to have mesh entities for all defined mesh entity ranks. In the finite element mesh example mesh entity ranks can be defined for nodes,

edges, faces, and element; but the mesh may only contain node and element mesh entities. A computational mesh is not required to have mesh entities with contiguous identifiers. As such the space for unique identification (rank$\times$identifier) is sparse.

A mesh entity's unique identification is persistent for the lifetime of that entity; *i.e.*, the rank and identifier of an entity cannot be changed. Given a mesh entity, its rank and identifier shall be computed in constant time, independent of the number of entities or entity types. Furthermore, mesh entities are well-ordered under the following relation.

$$entity_a^J < entity_b^K \iff \begin{cases} J < K & \text{or} \\ J = K \text{ and } a < b. \end{cases} \tag{2}$$

The mesh bulk data may be queried for a mesh entity ($entity_a^J$) by its rank and identifier (J,a). This query operation shall have a computational complexity that is at worst logarithmic with respect to the number of mesh entities contained within the mesh bulk data and on that process. This performance bound is acceptable because most computations on mesh bulk data iterate over selected mesh entities as opposed to performing these random access queries.

2.3 Mesh Part Membership and Efficient Selective Iteration

A mesh entity is a member of one or more mesh parts. Whereas the mesh entity's rank and identifier are persistent for the lifetime of that entity, the part membership of an entity may be changed. The *state of a mesh entity* is defined to be the set of application-defined parts for which it is a member. Put another way, the state of a mesh entity is the intersection of the mesh parts in which the mesh entity resides.

A mesh entity may be queried for the mesh parts for which that mesh entity is a member. This query operation shall have a computational complexity that is constant time, independent of the number of mesh entities or number of mesh parts.

An application's computation may operate on a collection of mesh entities of a selected mesh entity type and are members (or are not members) of a selected set of mesh parts. For example, in the fluid-in-a-pipe illustrative problem (Figure 1), a computation could compute on each node that is a member of the fluid part but is not a member of the pipe part.

An inefficient approach to iterate selected mesh entities is to iterate every mesh entity, query each each mesh entity for its mesh part memberships, and compute on those mesh entities which are members of the selected mesh part. In this example, selective iteration can be further complicated, and increasingly inefficient, when computing on each fluid node exactly once. Such an inefficient iteration would require:

- iterating every element of the mesh,

- querying if that element is a member of the fluid part, and, if so,
- querying that element for each of its nodes,
- somehow detecting and avoiding nodes on the fluid/pipe interface, and
- somehow detecting and avoiding reduntantly computing on nodes shared by two or more fluid elements.

Efficient selective iteration is achieved by inducing mesh part membership from mesh entity relations (Section 4.3) and grouping mesh entities into homogeneous subsets (Section 6). The just-the-fluid-nodes selective iteration example is simplified by inducing nodes' memberships in the fluid part from element-node relationships. This allows the fluid nodes to be iterated without having to iterate the fluid elements. Selective iteration in this example is made more efficient by grouping together nodes which are members of the fluid part. This allows selective iteration to query groups of nodes for their part membership, as opposed to querying individual nodes. Details of the conceptual design (a.k.a. domain model) for efficient selective iteration are developed in Section 3 through Section 6.

Selective iteration patterns include the following. Visit each entity of a given entity type exactly once. Visit each entity of a given entity type and member of a given mesh part exactly once. Visit each entity of a given entity type and member of a union, difference, or intersection of mesh parts exactly once.

3 Mesh Entity Relation

A *mesh entity relation* is a directed relationship between two entities contained within the same mesh, $entity_a^J \rightarrow entity_b^K$. The relationship is directed from the mesh entity with the greater rank to the mesh entity with the lesser rank. Thus, mesh entities of the equal mesh entity rank cannot have a relation, as per Equation (3).

$$entity_a^J \rightarrow entity_b^K \;\Rightarrow\; J > K \quad \text{therefore} \quad J \leq K \;\Rightarrow\; \nexists\, entity_a^J \rightarrow entity_b^K . \tag{3}$$

A mesh entity relation consists of the two mesh entities and a *relation attribute*,

$$Relation \equiv \left(entity_a^J \, , \, entity_b^K \, , \, attribute \right) , \tag{4}$$

where the *attribute* is a non-negative integer value. The mesh entity with the greater rank is referred to as the *from-entity* and the mesh entity with the lesser rank is referred to as the *to-entity*. An implementation for mesh entity relations may explicitly provide both forward relations (*e.g.*, element→node) and backward relations (*e.g.*, node→element). In such an implementation the *direction* of the relation is understood to be *from* the higher ranking entity *to* the lower ranking entity. Thus when a computational mesh implementation provides node→element data it is defined to be a mesh entity relation directed from the element to the node.

An entity may be a member of an arbitrary number of relations. For example, the hexahedral element in Figure 5 is the from-member of eight relations, one for each vertex-associated node. In this example, the eight element-to-node relations have relation attribute values of $[0..7]$, which define a convention for identifying vertices.

3.1 Uniqueness Condition

Higher ranking mesh entities are composed from, or defined by, lower ranking mesh entities (Section 2.1). Well-defined composition relatonships requires the uniqueness condition defined by Equation 5.

$$\begin{aligned} &\text{If} \quad \exists \left(entity_a^J \, , \, entity_b^K \, , \, d \right) , \exists \left(entity_a^J \, , \, entity_c^K \, , \, d \right) , \text{ and } J > K \\ &\text{then} \quad entity_b^K = entity_c^K \text{ and } b = c \, . \end{aligned} \tag{5}$$

This uniqueness condition requires that the combination of a from-entity, to-entity rank, and relation attribute $(entity_a^J \, , \, K \, , \, d)$ defines at most one mesh entity relation. In the example of Figure 5, each of the eight element-to-node relations are required to have a unique relation attribute value. However, if this element had an element-to-face relation then such

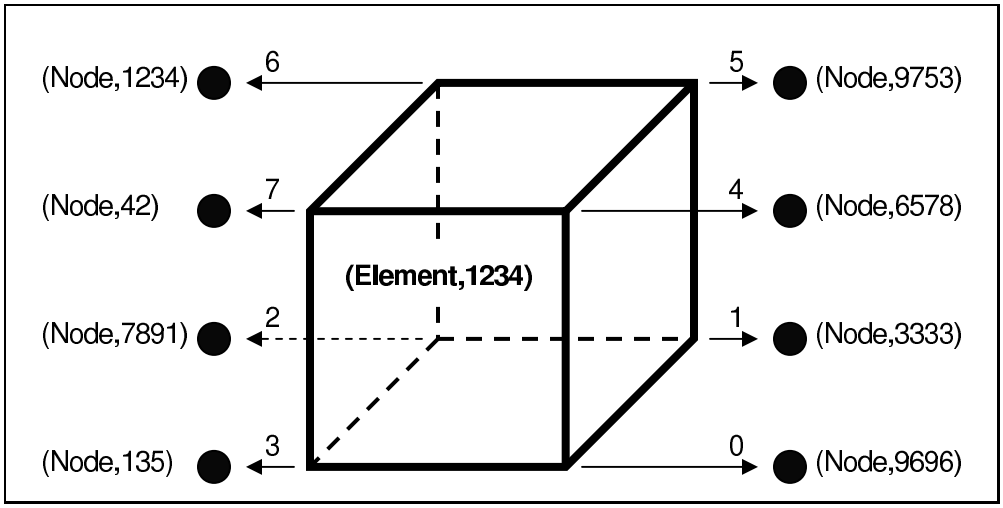


Figure 5. Example of mesh entity relations from a hexahedral element to eight nodes associated with the element's vertices

a relation is permitted to have an attribute value of "1" bacause the rank of a face is different than the rank of a node.

This uniqueness condition does **not** hold for the combination of a to-entity, from-entity rank, and relation attribute. For example, a node is permitted to be the to-entity of more than one element-to-node relations with an attribute value of "1". This is permitted because from-entities are composed from, or defined by, the to-entities of their relations and not vice-versa.

3.2 Querying and Ordering

Computations on a mesh entity often iterate over a given mesh entity's relations (Equation 7). For example, computations on a given face ($entity_a^2$) could iterate over

- face→node relations ($entity_a^2 \rightarrow entity^0$),
- face→edge relations ($entity_a^2 \rightarrow entity^1$), or
- element→face relations ($entity^3 \rightarrow entity_a^2$).

The ordering of mesh entity relations (Equation 9) is defined such that each of these iteration scenarios accesses a span of the well-ordered mesh entity relations $Relation\left(entity_a^2\right)$.

It is frequently necessary to refer to subsets of relations for which a given mesh entity is a member. For such relation subsets are compactly described in Equation 6 via the $\rightarrow$ operator. Let R be the set of all relations in a mesh, then the following expressions for

subsets of R are defined.

$$\begin{aligned}
entity_a^J \rightarrow entity^K &\equiv \left\{ \left(entity_a^J \, , \, entity_b^K \, , \, x \right) \in R \, ; \forall \, b,x \right\} \, : \, J > K \\
entity_a^J \rightarrow entity &\equiv \left\{ \left(entity_a^J \, , \, entity_b^K \, , \, x \right) \in R \, ; \forall \, K,b,x \, : \, J > K \right\} \\
entity^K \rightarrow entity_a^J &\equiv \left\{ \left(entity_b^K \, , \, entity_a^J \, , \, x \right) \in R \, ; \forall \, b,x \right\} \, : \, K > J, \text{ and} \\
entity \rightarrow entity_a^J &\equiv \left\{ \left(entity_b^K \, , \, entity_a^J \, , \, x \right) \in R \, ; \forall \, K,b,x \, : \, K > J \right\}.
\end{aligned} \tag{6}$$

For example, the element ($rank = 3$) $\rightarrow$ node ($rank = 0$) relations for element a are compactly expressed as $entity_a^3 \rightarrow entity^0$.

Given $entity_a^J$, the complete subset of mesh entity relations for which that entity is a from-member or a to-member is identified by $Relations\left(entity_a^J\right)$, where

$$\begin{aligned}
Relations\left(entity_a^J\right) &\equiv \left(entity_a^J \rightarrow entity\right) \bigcup \left(entity \rightarrow entity_a^J\right) \\
&= \left\{ \left(entity_a^J \, , \, entity_b^K \, , \, x \right) \in R \, ; \forall \, K,b,x \, : \, J > K \right\} \bigcup \\
&\quad \left\{ \left(entity_b^K \, , \, entity_a^J \, , \, x \right) \in R \, ; \forall \, K,b,x \, : \, K > J \right\}.
\end{aligned} \tag{7}$$

The mesh bulk data for $Relations\left(entity_a^J\right)$ subset can be compactly represented as

$$Relations\left(entity_a^J\right) \rightarrow \left\{ \left(entity_b^K, x\right) : \left(entity_a^J, entity_b^K, x\right) \text{ or } \left(entity_b^K, entity_a^J, x\right) \in R \right\} \tag{8}$$

where the members of this compact representation are well-ordered

$$\left(entity_b^K, x\right) < \left(entity_c^L, y\right) \Leftrightarrow \begin{cases} K < L & \text{or} \\ K = L \, , \, x < y & \text{or} \\ K = L \, , \, x = y \, , \, b < c. \end{cases} \tag{9}$$

A query operation to access this subset shall have a constant time complexity; *i.e.*, no search operations shall be required. The results of this query operation shall support iteration conformal to the defined ordering.

3.3 Closure of an Entity

A mesh's collection of entities and relations forms a directed acyclic graph (DAG) where entities are nodes in the graph and the directed relations are the edges. This graph is guaranteed to be acyclic due to relations being directed from entities of greater rank to entities of lesser rank, and disallowing relations between entities of the same rank. Thus the traversal semantics and algorithms applicable to a DAG are applicable to a mesh.

A computation on an entity commonly operates on data associated with that entity and those entities obtained by following all relations directed away from the entity. For example, an element computation typically operates on data from the element and its connected nodes.

The *closure of an entity* is the collection of entities and relations reachable from that entity. For example, the closure of an element consists of the element and any faces, edges, nodes, and relations that lie in a path originating from the element. A contrasting example is the closure of an edge includes its nodes, but not the higher-ranking faces or elements connected to that edge since all paths from edges must point to entities of lower types.

The closure of an entity is guaranteed to be well-defined and bounded. The concept of the closure of an entity is critical to generating a well-defined parallel-distribution of entities. For example, if an entity is to be computed on by a given process then the closure of that entity must also reside on that process for efficiency.

3.4 Customary Finite Element Topological Relations

Mesh entity relations are used to implement finite-element topological relations, as illustrated in Figure 5. These relations are identified by defining a specific relation attribute value for each defined topological relationship. For example, the vertices of the hexahedral element in Figure 5 are uniquely identified by the relation attribute values of $[0..7]$

Topological relations may not be fully populated. For example, a hexahedral element entity can have relations to six quadrilateral face entities; however, only one such face entity and corresponding relation might exist. In this scenario, it is useful for the relation attribute to identify which element-face in the element's local face-numbering convention is denoted by the element$\rightarrow$face relation.

Cell Topology: The family of Sierra Mechanics applications use element entities that conform to the standard element types defined in the ExodusII manual [8]. These ExodusII standard element types define a local numbering convention for element-node and element-face relationships for beam, triangle, quadrilateral, tetrahedral, pyramid, wedge, and hexahedral element entities with various numbers of nodes. The ExodusII standard element type numbering conventions describe connections from a higher dimensional topological entity to its lower dimensional topological entities that are associated with its faces, edges, or vertices. This concept of a standard description for element topological relationships includes:

- local numbering conventions for topologically related entities,
- descriptions of the expected topologically related entities (*e.g.*, a pyramid is expected to have one quadrilateral face and four triangular faces), and
- orientation-mappings between the topologically related entities (*e.g.*, the vertices of local face #1 of a pyramid are defined by the pyramids local vertices #1, #2, and #5).

3.5 Examples of Non-topological Relations

Non-topological relations have been used between mesh entities in Sierra Mechanics application codes. These include

- constraint entity relations with its constrained entities, and
- relations between level set surface entities and volume mesh entities.

The flexibility and extensibility of the mesh entity relation concept support research and development of these and other discretization approaches.

4 Mesh Part

A *mesh part* defines a subset of an application's mesh entities. A part may be used to reflect the physics modeled, discretization methodology, solution algorithm, meshing artifacts, or other application requirements.

4.1 Identifiers and Attributes

A mesh part has an unique text name identifier which is specified by the application. This text name is intended to support text input and output by the application, *e.g.*, parsing, logging, and error reporting. The text name is not intended for referencing a mesh part within application computations. Such undesired text-based references will lead to text-based searches within the application's computations, resulting in unnecessarily degraded performance.

A mesh part has a unique non-negative integer identifier which is internally generated by the mesh meta data. This internal integer identifier is intended to support fast referencing and ordering of mesh parts. The internal integer identifier is also intended to support efficient communication of mesh part information among distributed memory processes. Given the need for efficient referencing, querying a mesh part from its integer identifier shall have constant time complexity.

An application may specify that a mesh part is intended to define a subset of mesh entities of a given mesh entity rank. For example, an "element block" mesh part is intended to define a collection of elements.

$$\begin{aligned} Part_A &\equiv \text{mesh part identified by } A \\ Part_A^J &\equiv \text{mesh part intended for mesh entities of rank } J \text{ and identified by } A \end{aligned} \tag{10}$$

4.2 Set Hierarchy

Two mesh parts within the same mesh can be given a directed subset relationship that defines a partial order on the mesh parts. If the relationship $Part_A \xrightarrow{\supset} Part_B$ is defined then by directedness the reverse relationship cannot be defined. The set hierarchy can be used to aggregate similar parts, or partition a given part into subsets with unique properties. For example, a collection of independent boundary condition parts may be aggregated with an "all boundary conditions" superset part.

4.3 Induced Entity Membership

A mesh entity may be an *explicit member* or an *induced member* of a mesh part. A mesh entity can be explicitly inserted into or removed from a mesh part. A mesh entity's membership in a part may also be induced through its relationship to a higher rank mesh entity. There are two methods of inducing entity membership in a part. The first method induces entity membership simply based on entity relations. The second method induces entity membership based on a subset of entity relations selected by a *stencil* (Equation 12).

4.3.1 Relation-based membership

When a mesh part has a specified entity rank ($Part_A^J$) then only mesh entities of the same entity rank J may be explicitly added as members to that mesh part. If a mesh entity is an *explicit member* of such a mesh part, $entity_a^J \in Part_A^J$, and that mesh entity ($entity_a^J$) is the from-entity of a relation, then the to-entity of that relation is an *induced member* of that mesh part.

$$\begin{array}{lll} \text{Given} & \text{a relation } \left(entity_a^J \, , \, entity_b^K \, , \, x \right) \, : \, J > K \quad \text{and} & \\ & entity_a^J \in Part_A^J \quad \text{via explicit membership} & (11) \\ \text{then} & entity_b^K \in Part_A^J \quad \text{via induced membership.} & \end{array}$$

Induced membership only occurs in the presence of a mesh entity relation; *i.e.*, induced membership is not transitive. For example, if a mesh has both element-to-face and face-to-edge relations, but does not have element-to-edge relations, then the edges in the element's closure (via element-to-face-to-edge) are **not** induced members.

Recall the selective iteration example of iterating all nodes in the fluid mesh part introduced in Section 2.3, Mesh Part Membership and Efficient Selective Iteration. In this example an application would designate that the fluid mesh part is intended for elements, $Part^3_{\text{Fluid}}$. Through this designation all nodes of the fluid elements are induced to be members of the fluid part. As such these fluid nodes can be selectively iterated without having to iterate the fluid elements and traverse the element-to-node relations.

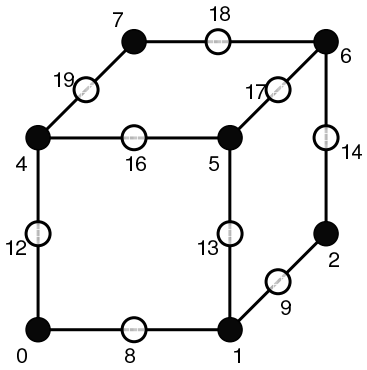
4.3.2 Stencil-based membership: part relation stencil

A *part relation stencil* is a means of selecting a subset of mesh entity relations through which part membership is induced. Recall the example of a fluid element introduced in Figure 2. In the fluid computation the vertex nodes and mid-edge nodes are treated differently; the vertex nodes have both velocity and pressure field data while the mid-edge nodes have only velocity field data. In this example a stencil is used to select those element-to-node relations that are associated with the element's vertices.

A *stencil* is a function that maps a mesh entity relation's from-entity rank, to-entity rank, and relation attribute to an integer value (Equation 12). If the mapped value is non-negative then the associated relation is accepted. A negative value indicates that the associated relation is rejected.

$$\textit{stencil} : (J , K , x) \mapsto \mathbb{Z} \quad \left\{ \begin{array}{lcl} J & \equiv & \text{a relation's from-entity rank} \\ K & \equiv & \text{a relation's to-entity rank} \\ x & \equiv & \text{a relation's attribute} \end{array} \right. \tag{12}$$

The element in Figure 6 has twenty element-to-node relations, each with a specific relation attribute value in the range $[0..19]$. In this example the eight element-to-node relations associated with the element's vertices have attribute values in the range $[0..7]$. The pressure stencil in Figure 6 selects this element's vertex-associated element-to-node relations. The particular non-negative return value of the stencil is not important for part relation stencils; however, this return value is important to field data relation stencils (Section 5.4).



$$\text{pressure\_stencil}(J,K,x) = \left\{ \begin{array}{rl} x & : J = \text{element},\ K = \text{node},\ x < 8 \\ -1 & : \text{otherwise} \end{array} \right.$$

Figure 6. Example element-to-node relation attribute values and stencil functions for the illustrative fluid element

A part relation stencil is the combination of a from-part, to-part, and stencil function. When the from-entity of a relation is explicitly a member of the from-part, and the relation is accepted by the stencil function, then the to-entity of that relation is induced to be a

member of the to-part. This part relation stencil rule is denoted by Equation 13.

$$
\begin{array}{lll}
\text{Given} & \text{a part relation stencil} = \left(\mathit{Part}_{\text{from}} \, , \, \mathit{Part}_{\text{to}} \, , \, \mathit{stencil} \right) & \text{and} \\
 & \text{a relation} = \left(\mathit{entity}_a^J \, , \, \mathit{entity}_b^K \, , \, x \right) \, : \, J > K & \text{and} \\
 & 0 \leq \mathit{stencil}(J,K,x) & \text{and} \\
 & \mathit{entity}_a^J \in \mathit{Part}_{\text{from}} \quad \text{via explicit membership} & \\
\text{then} & \mathit{entity}_b^K \in \mathit{Part}_{\text{to}} \quad \text{via induced membership.} &
\end{array}
\tag{13}
$$

Mesh entities are prohibited from being explicit members of the to-part of part relation stencil ($\mathit{Part}_{\text{to}}$ in Equation 13). This additional rule is imposed so that membership in the to-part is well-defined. Well-defined rules for induced part membership are necessary for manageable modification of mesh bulk data (Section 9).

Recall that in the illustrative flow-in-a-pipe problem (Figure 1) only some of the nodes in the fluid part have pressure field data (Figure 2). These nodes become induced members of a fluid-pressure-node mesh part ($\mathit{Part}_{\text{press}}$) by

- adding the fluid elements as explicit members of the fluid-part ($\mathit{Part}_{\text{fluid}}$),
- using the element-to-node relation attributes defined in Figure 6, and
- defining a mesh part stencil $\left(\mathit{Part}_{\text{fluid}}, \mathit{Part}_{\text{press}}, \text{pressure\_stencil}\right)$ with the pressure stencil defined in Figure 6.

5 Mesh Field

A computational mesh is *field-centric*—the purpose of entities, relationships, and parts is to provide a foundation upon which computational field data is defined and used. A field specification has three components: (1) field type, (2) field declaration, and (3) field restrictions. Field data are automatically created for mesh entities in compliance with the field specification.

5.1 Field Type and Declaration

A *field type* is an application-defined multidimensional array of a numerical data type (*e.g.*, int, float, double). The number of dimensions of the array and intent of each dimension are components of the field type. For example, an array may be dimensioned by the number of spatial ordinates, the number of basis function ordinates, and the number of integration rule ordinates:

```
REAL*8 ArrayType( nSpace , nBasis , nIntegration )    ! FORTRAN notation

double ArrayType[ nIntegration ][ nBasis ][ nSpace ]  // C or C++ notation
```

A *field declaration* defines a specific computational field of a specified field type within a computational mesh. For example, the vector fields of coordinates, velocity, and acceleration are three distinct field declarations all having the same field type. A field declaration has an identity (or handle) through which the field is referenced in constant time-complexity. A field declaration also has a text name to support input or output references to that field.

A potential use case is for an array type to be the auto-differentiation types implemented by SNL's Sacado package within Trilinos [**?**]. If such complicated data types are supported then they must support the "plain old data" [4] semantics for simple creation, assignment, and destruction operations.

5.2 Field Data

Field data values are associated with, or defined by, the pairing of a field declaration with an entity; e.g., (field-declaration , $entity_a^I$) $\rightarrow$ field-data. Not all entities will have field data for every field declaration. For example, in the illustrative flow-in-a-pipe problem (Figure 1) only the nodes with relations to fluid elements will have field data for the fluid velocity and pressure. The *field restriction* defines whether or not field data exists for a given (field-declaration , $entity_a^I$) pair.

Field data are accessed in different ways, depending upon the computation being performed. Examples of common field data access patterns include the following.

- Access all field data for a collection of fields without knowledge of the associated entities. Uses cases include computing a norm of a field, adding two fields into a third field or performing a simple kinematic update on acceleration, velocity, and coordinate fields during a time step.
- Access field data for a collection of fields and collection of entities within a selected subset of the mesh. Use cases are similar to the previous example except on a part-defined subset of the mesh. For instance, applying a boundary condition.
- Access field data associated with the collection of mesh entities related to a given mesh entity ($entity_a^I \rightarrow entity$). For example, access field data associated with nodes of a given element. Use cases typically traverse well-defined relationships (e.g., defined by a cell topology) and read or update field data on the to-entity of the relationship. Such field data access patterns have been referred to as "gather" or "scatter add."

The performance of computations is sensitive to the layout of field data associated with collections of entities and fields. This performance sensitivity is pronounced for linear algebra operations. Array-like contiguous field data layouts provide better performance than other non-contiguous field data layouts. Thus, array-like contiguous field data structures are used as much as feasible. However, such data structures can be complicated by the heterogeneous nature of the mesh and the field data layout.

5.3 Field Restriction for Heterogeneous Field Data

A *field restriction* is a rule which defines both the existence of field data and the dimensions of the field data array. The dimensions of the field data are dependent upon the entity with which the field is paired. For example, a field may have one of its dimensions associated with the integration points on a mesh entity, which can vary with the mesh entity's topology and approximation order (*e.g.*, linear, quadratic).

A field restriction maps a specific field declaration ($Field_X$), mesh entity rank (J), and mesh part ($Part_A$) to an array dimension. Field data will exist for a given field declaration and mesh entity ($Field_X, entity_b^K$) if-and-only-if a field restriction exists to relate the field declaration and mesh entity as per Equation 15.

$$\begin{aligned} &\text{Field-Restriction} \equiv (Field_X, J, Part_A) \rightarrow ArrayDim_{X,J,A} \\ &\text{where } ArrayDim_{X,J,A} = (N_1, N_2, \ldots) \text{ or } [\ldots][N_2][N_1] \,. \end{aligned} \tag{14}$$

$$\begin{aligned} &\text{The pair } \left(Field_X, entity_b^K\right) \rightarrow FieldData\left[ArrayDim_{X,K,A}\right] \iff \\ &\exists\ (Field_X, K, Part_A) \rightarrow ArrayDim_{X,K,A} \text{ such that } entity_b^K \in Part_A \ . \end{aligned} \tag{15}$$

A direct consequence is that if no such field restriction exists then that pairing of a field declaration and mesh entity will not have field data.

Multiple field restrictions may exist for a given field declaration. Some of these field restrictions may specify different array dimensions. In this situation, the consistency condition of Equation 16 must be enforced.

$$\begin{aligned} \text{If} \quad & \exists\ (Field_X, K, Part_A) \rightarrow ArrayDim_{X,K,A} && \text{and} \\ & \exists\ (Field_X, K, Part_B) \rightarrow ArrayDim_{X,K,B} && \text{and} \\ & \exists\ entity_b^K \in Part_A \cap Part_B \\ \text{then} \quad & ArrayDim_{X,K,A} = ArrayDim_{X,K,B} \ \text{ for consistency.} \end{aligned} \tag{16}$$

5.4 Field Data Relation Stencil

Computations can access field data associated with mesh entities in the set $entity_a^I \rightarrow entity$ by iterating the relations of this set and querying field data from each of the to-entities. The overhead of this common field data access pattern can be amortized by saving the results of the relation iteration and field data query.

A *field data relation stencil* specifies that the results of this field data access pattern shall be maintained in an array of pointers to field data; *e.g.*, `double * pointers[N]` in C or C++ notation. The pointer field data (`pointers`) is associated with the from-entity of the relations. Field data associated with related entities can then be accessed through the pointer field data without the overhead of iterating over relations and retrieving field data from related entities. The runtime performance improvement can be significant, at the expense consuming more memory for the pointer field data.

A *field data relation stencil* formally defines a relationship between a pointer-valued field $Field_{PX}$ and another field $Field_X$. For example, an element may have a pointer-valued field with values that point to the coordinate field associated with its nodes. Such a pointer-to-node-coordinates field allows computations on that element to directly access the nodal coordinates without having to traverse the element-to-node relations. The stencil function use here has the identical interface and functionality as the stencil function used for part

relation stencils (Section 4.3). This relationship is defined by Equation 17.

$$
\begin{array}{lll}
\text{Given} & \text{a field relation stencil } \left(\text{pointer } \mathit{Field}_{PX}, \text{value } \mathit{Field}_{X}, \mathit{stencil}\right) , & \\
\text{if} & \exists\ (\mathit{Field}_{PX}, J, \mathit{Part}_{A}) \rightarrow (N) & \text{and} \\
 & \exists\ (\mathit{Field}_{X}, K, \mathit{Part}_{B}) \rightarrow \mathit{ArrayDim}_{X,K,B} & \text{and} \\
 & \exists\, \mathit{Relation} = \left(\mathit{entity}_{a}^{J}, \mathit{entity}_{b}^{K}, \mathit{attr}\right) & \qquad (17) \\
\text{such that} & \mathit{entity}_{a}^{J} \in \mathit{Part}_{A} \ \text{ and } \ \mathit{entity}_{b}^{K} \in \mathit{Part}_{B} & \text{and} \\
 & i = \mathit{stencil}\,(J, K, \mathit{attr}) \ \ , \ \ 0 \leq i < N & \\
\text{then} & \text{pointer}\left(\mathit{Field}_{PX}, \mathit{entity}_{a}^{J}\right)[i] \ \ \mathit{references} \ \ \text{value}\left(\mathit{Field}_{X}, \mathit{entity}_{b}^{K}\right). &
\end{array}
$$

6 Mesh Bucket: Homogeneous Chunk of Field Data

Field data can be defined heterogeneously (Section 5.3) over a mesh such that field data may not exist for a given mesh entity, and if it does exist, the dimensions of the field data array can vary between mesh entities. However, there will always exist subsets of mesh entities which are of the same entity rank and are members of the same mesh parts. The mesh entities within such a homogeneous subset will have the same field data, and that field data will have the same array dimensions.

Consider a problem domain in with three mesh parts: A, B, and C (Figure 7). In this example, there are seven possible homogeneous subsets in which an entity may be a member.

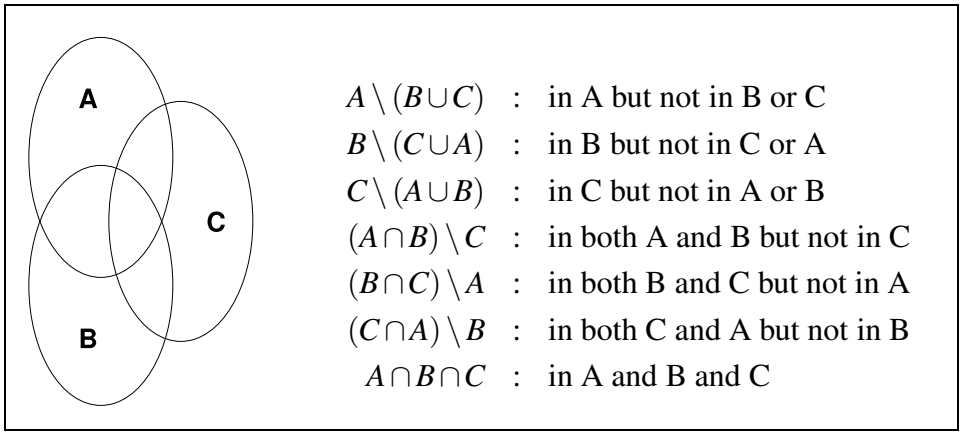


Figure 7. Example of potential homogeneous subsets from problem domain with three mesh parts.

6.1 Buckets, Entities, Parts, and a Mesh

A *mesh bucket* is a block of contiguous memory that contains all of the field data associated with a homogeneous subset of mesh entities. The mesh entities are homogeneous in that they are of the same entity rank and are members of the same subset of mesh parts. Each mesh entity is associated with exactly one mesh bucket. Thus, the collection of a mesh's buckets defines a partitioning of the mesh's bulk data. The relationships between mesh, mesh parts, mesh entities, and buckets are illustrated in Figure 8.

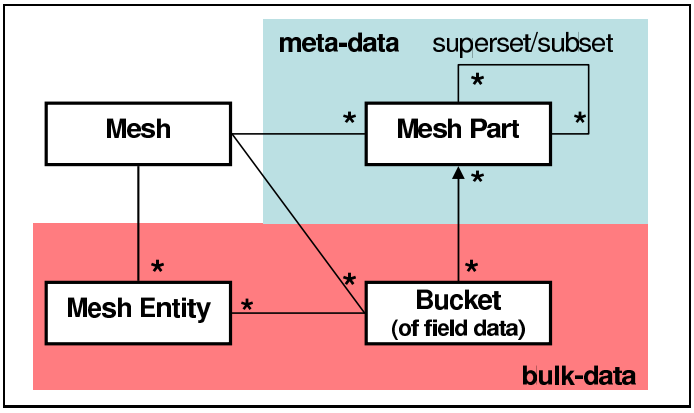


Figure 8. Illustration of relationships between buckets, entities, parts, and mesh.

6.2 Iteration and Separation of Concerns

Partitioning a heterogeneous collection of mesh entities into homogeneous subsets encourages a separation of concerns leading to better performance. First, all field data associated with a homogeneous subset of entities are maintained in contiguous multidimensional arrays, with the longest-striding dimension corresponding to the number of entities in the subset. This allows a computational kernel to access many entities' field data through a simple loop over the number-of-entities dimension. Second, the logic which selects mesh entities for computations (recall Section 2.3 on mesh entity iteration) can be applied to choose a bucket of mesh entities, and does not have to be repeated for each mesh entity within that bucket's homogeneous subset. Third, this abstraction encourages a separation of concerns between entity selection logic and computational kernels—allowing the efficient use of dense computational kernels, such as basic linear algebra subprograms [12]. This separation of concerns is illustrated in Algorithm 1.

```
foreach Bucket in Bulk Data do
    if Bucket is selected then
        get number of entities N associated with Bucket
        get reference to field data array A in Bucket
        get reference to field data array B in Bucket
        ⋮
        call ComputationalKernel(N,A,B,...)
    end
end
```

Algorithm 1: Algorithm for bucket selection and entity field data array iteration.

This algorithm has three application-supplied operations: selecting buckets, getting ref-

erences to selected field data from a selected bucket, and invoking a computational kernel on selected field data. The two operations of selecting a bucket and getting references to selected field data within a selected bucket are overhead operations (*i.e.*, non-computational kernel operations) which would not be necessary for a completely homogeneous problem. Two assumptions are made regarding this overhead. First, that an application will, at some point in its lifecycle, be required to cope with heterogeneity. Second, that these overhead operations will require signficantly less time compared to the computational kernel's operations on the bucket's field data.

6.3 Performance Considerations

The possible number of buckets grows factorially with the number of mesh parts. As such, it is prudent to create only non-empty homogeneous subsets and corresponding buckets whenever possible.

Fragmentation of complex meshes: If the number of mesh entities is large compared to the number of mesh parts, then there will be (on average) a large number of entities in each bucket. This is the assumed HPC application use case; *e.g.*, millions of entities but only hundreds of parts. If the number of mesh entities is relatively small compared to the number of parts then there will be a small number of mesh entities in each bucket.

In large (many mesh entities) and complex (many mesh parts) problem domains the number of mesh entities in each mesh part varies significantly. For example, some mesh parts have millions of mesh entities while other mesh parts have tens of mesh entities.

Cardinality of buckets: The number of mesh entities associated with each homogeneous (*i.e.*, the cardinality of the homogeneous subset) is highly variable and problem dependent. The cardinality of a homogeneous subset will impact runtime performance. If the cardinality is small then the runtime overhead (selecting buckets and getting references to selected field data) is proportionately larger compared to the runtime of the computational kernel. However, if the cardinality becomes large then the runtime and memory overhead for adding/removing mesh entities to/from a buckets can increase due to memory management operations. It is this too-small versus too-large tradeoff that led to allowing a homogeneous subset to be partitioned into multiple buckets, where an application defines a maximum cardinality for buckets.

7 Parallel Distribution

Mesh bulk data is distributed among the processes of a distributed memory parallel environment. A domain decomposition algorithm is typically used to assign mesh entities to processes. In order to support computations on a mesh entity, the mesh entity and its closure must be present on the process performing the computation. This requirement necessarily causes some mesh entities and their field data to be duplicated on more than one process.

Consider a parallel domain decomposition of the illustrative problem in Figure 9 that assigns elements to processes. The closures of these elements contain nodes which, if they are on the boundary of the parallel domain decomposition, will reside in two or more processes. These nodes are *shared* by the processes in which the nodes reside.

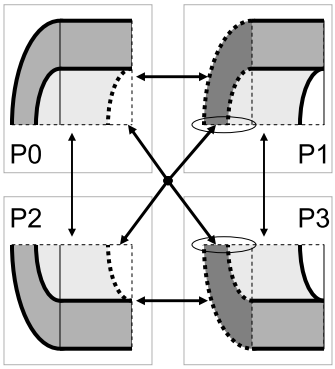


Figure 9. Mesh entity sharing in illustrative multi-physics problem parallel domain decomposition.

7.1 Parallel Ownership and Sharing

When a mesh entity resides on more than one process then an owner is selected from among those processes. A mesh entity is *owned* by exactly one of the processes in which it resides. This owner process establishes the correct field data and state when there is a discrepancy among processes. A global reduction, such as a dot-product, includes only contributions from owning processes. The collection of process-owned subsets of a mesh define a partitioning of a mesh's entities.

$$\begin{aligned} \text{Owned}_P &\equiv \text{subset of entities owned by process } P \\ \bigcup_P \text{Owned}_P &= \text{all mesh entities} \\ \text{Owned}_P \bigcap \text{Owned}_Q &= \emptyset \ \forall \ P \neq Q. \end{aligned} \tag{18}$$

The closure of a subset of owned mesh entities is typically known as a *subdomain* of a parallel domain decomposition. This subdomain contains the mesh entities and associated field data that are most often required for computations performed by a process.

$$\overline{\text{Owned}_P} \equiv \begin{cases} \text{closure of all owned mesh entities on process } P\text{;} \\ \textit{i.e.}\text{, mesh entities used by process } P\text{.} \end{cases} \tag{19}$$

The closure of an owned mesh entity is required to exist within the process that owns the entity. Mesh entities in the closure of two different processes' owned subsets are *shared* by those processes.

$$\text{Shared}_{P,Q} \equiv \overline{\text{Owned}_P} \bigcap \overline{\text{Owned}_Q} \ \forall \ P \neq Q \tag{20}$$

Note that a mesh entity may be in the closure of several processes' owned subsets, as illustrated in Figure 9, and thus have multi-way sharing.

Some computations require communication and synchronization of field data associated with shared mesh entities. For example, a computation that generates and accumulates partial sums on each process must sum results among processes that share mesh entities as in the "swap-add" operation. This communication operation requires the following communication list, where process Q sends / receives field data for $entity_a^J$ to / from process P.

$$\text{SharedComm}_Q \equiv \{(entity_a^J, P) : entity_a^J \in \text{Shared}_{P,Q} \ \forall \ P \neq Q\} \tag{21}$$

Members of SharedComm_Q communication list are fully ordered by entity rank, then by entity identifier, and finally by process rank. This ordering groups sharing data by mesh entity and supports efficient querying of which process' share a given mesh entity.

Reduction operations: A dot product computation must visit field data for each mesh entity exactly once to generate the correct result. Thus, iteration of mesh entities and mesh buckets must include selection of mesh entities which are owned by the local process. This selective iteration is supported by defining an owned mesh part which is composed of the owned mesh entities as members of that mesh part. Selective iteration is then accomplished by selecting buckets which are subsets of the owned mesh part as well as any other application-defined mesh parts.

7.2 Parallel Ghosting

A mesh entity that resides in a process and is not in the closure of the process' owned subset is termed a *ghost* mesh entity.

$$\begin{aligned} \overline{\text{Owned}_P} \bigcup \text{Ghosted}_P &\equiv \text{all entities existing in process } P \\ \overline{\text{Owned}_P} \bigcap \text{Ghosted}_P &\equiv \emptyset \end{aligned} \tag{22}$$

Ghost entities are typically needed by computations that gather field data from an owned entity's neighbors. For example, a computation applied to an owned element may gather field data from other elements that share a side with the owned element (*e.g.*, an upwinding computation). These side-neighbor elements may be owned by another process and not shared by the local process. Thus, they may be parallel-ghosted on the local process to improve performance.

Field data must be communicated from an owned mesh entity to the ghosted parallel-replication of that entity before the field data can be gathered from the ghosted mesh entity. This communication operation requires the following communication lists, where $(entity \rightarrow P)$ means the entity is being communicated to P.

$$\begin{aligned} \text{GhostCommSend}_Q &= \{(entity \rightarrow P) : entity \in \text{Ghosted}_P \bigcap \text{Owned}_Q \; \forall P \neq Q\} \\ \text{GhostCommReceive}_Q &= \{(entity \leftarrow P) : entity \in \text{Ghosted}_Q \bigcap \text{Owned}_P \; \forall P \neq Q\}. \end{aligned} \tag{23}$$

Note that a ghosted mesh entity will appear exactly once in a process' GhostCommReceive list, but may appear many times in a process' GhostCommSend list.

Multiple independent computations may require their own particular subsets of parallel-ghosted mesh entities. These computation-specific $\{\text{Ghosted}_P\}_\Phi$ subsets are mesh bulk data that is created, owned, modified, and destroyed by an application.

7.3 Parallel Consistency

When a mesh entity is duplicated on two or more processes (shared or ghosted) then the parallel consistency of that mesh entity is a concern. A mesh entity is *parallel consistent* if it meets the following three consistency criteria.

- **Consistency of duplicated mesh entities' state**—each shared mesh entity is a member of the same application-defined mesh parts on each process. This consistency condition does not include the owned mesh part because a shared mesh entity will be a member of this owned mesh part on exactly one process.

- **Consistency of duplicated mesh entities' relations**—each shared mesh $entity_a^J$ has the same $entity_a^J \rightarrow entity$ relations on each process. This consistency condition cannot include the $entity \rightarrow entity_a^J$ relations as such a condition would consequently lead to duplication of the entire mesh on every process.
- **Consistency of duplicated mesh entities' field data**—the field data values associated with each mesh entity are the same on each process. This last consistency concern is left up to the application. However, some "helper" communication operations for maintaining consistency is provided.

8 Mesh Meta Data Management

Mesh entity rank, mesh part, and mesh field specifications are grouped into a *mesh meta data* aggregate. This aggregate has the responsibility for creating, owning, modifying, and destroying mesh meta data. Thus, an application works through the mesh meta data aggregate to create, modify, and destroy mesh meta data.

A mesh meta data aggregate has two states: under construction and in use. When an application creates a mesh meta data aggregate it is in the under construction state. Mesh meta data specifications are created and modified while the aggregate is in the under construction state. After a mesh meta data aggregate is fully populated the application commits it to the in use state. Mesh meta data invariants are enforced while mesh meta data is created and modified, and verified when committed to the in use state. Once in the in use state it cannot be reverted to the under construction state.

Parallel duplication: Mesh meta data is assumed to be duplicated in all processes over which the computational mesh will exist. It is assumed that this requirement will not consume an unreasonable amount of the processes' memory. It is the responsibility of the application to construct the mesh meta data through the same sequence of creation and modification operations on each process. For performance efficiency, the consistency of mesh meta data on each process is not verified until after the construction process is completed.

9 Mesh Bulk Data Management

A *mesh bulk data aggregate* is composed of mesh entities and their part memberships, relations, field data, and parallel communication lists. This aggregate has the responsibility for creating, owning, modifying, and destroying mesh bulk data conforming to mesh meta data specifications. An application (1) creates a mesh bulk data aggregate, providing a mesh meta data aggregate in the consistent state, and (2) works through the mesh bulk data aggregate to create, modify, and destroy mesh entities and relations. The mesh bulk data aggregate automatically allocates and manages mesh field data according to the mesh meta data specifications provided by the application.

A mesh bulk data aggregate has two states: *modifiable* and *consistent*. When an application creates a mesh bulk data aggregate, with the application's mesh meta data aggregate, it is in the modifiable state. Mesh bulk data are created, modified, and destroyed while the aggregate is in the modifiable state. After a mesh bulk data aggregate is satisfactorily populated the application changes it to the consistent state. Mesh bulk data invariants (Section 1.3) are enforced on the local process while mesh bulk data are being created and modified, and verified when changed to the consistent state.

Parallel distribution: Mesh bulk data are distributed among parallel processes. Creation, modification, and deletion of mesh bulk data can affect parallel consistency. Parallel consistency is resolved and enforced when the application changes the mesh bulk data from the modifiable state to the consistent state. In this approach, requisite parallel communication can be performed as a parallel collective batch operation, which provides the greatest opportunity to reduce parallel communication overhead. In contrast, enforcing parallel consistency at each modification would have an unacceptable parallel communication overhead.

9.1 State of a Mesh Includes Ordering of its Data

Ordering of data (*e.g.*, mesh parts or mesh entities) is significant because computations frequently have an order-dependency. For example, floating point operations performed by a CPU are not associative — they are order-dependent. Thus an order-dependent computation performed on two meshes with the exact same mesh meta data and mesh bulk data will have different results if that data has a different ordering.

The *state* of a mesh is defined by its mesh meta data, mesh bulk data, **and** the ordering of that data. This definition is made to allow applications to perform order-dependent computations on a mesh with repeatable results. For example, an application which checkpoints (saves to permanent storage) a mesh and subsequently reloads that mesh can assume that its order-dependent computations to produce the same results as if the mesh had not been check-pointed and restarted.

Consistent ordering of mesh data is enforced when an application changes the mesh bulk data aggregate from the modifiable state to the consistent state. This state transition includes an operation to sort the mesh entities into a well-defined and consistent order. This ordering is **not** enforced when the mesh is in the modifiable state as other sources of inconsistencies will also be present, and enforcement during mesh modifications can introduce undesirable runtime overhead.

The ordering of mesh data is inherently dependent upon its parallel distribution. A different parallel distribution will result in a different ordering, which results in a different result from an order-dependent computation. Thus for a computation to be insensitive to the parallel distribution of a mesh it must also be order-independent, **or** that computation may iterate data in a prescribed order. Reordering or remapping of mesh data is a potential enhancement described in Section 11.3.

9.2 Runtime Modifications

An application may change the mesh bulk data from the consistent state to the modifiable state as many times as required. However, each subsequent change to the consistent state will have the overhead of resolving and enforcing parallel consistency. As such, it is strongly recommended that an application change that state of the mesh bulk data only when necessary, perform as many modifications as are needed, and then change the mesh bulk data to the consistent state. The expected sequence of operations performed by an application on its computational mesh are illustrated in Algorithm 2.

Runtime modifications to mesh bulk data often affect only a localized portion of the mesh. For example, a mesh adaptation algorithm may refine some tens of elements in a million-element mesh to reduce the spatial discretization error. In such a localized-modification scenario, the parallel consistency resolution operation, and other operations that update external dependencies on MeshBulkData (Algorithm 2), should be proportionately localized for better runtime performance.

9.3 Transaction Strategy for Modifications

Mesh bulk data modifications which occur between changing to the modifiable state and changing back to be in use constitute a *transaction* on the mesh bulk data. This state-change and transaction strategy, illustrated in Algorithm 2, is chosen to (1) support efficient enforcement of parallel consistency of distributed mesh bulk data, and (2) support coordination of modifications among components of an application. Such a coordination operation is reflected in the update external dependences on MeshBulkData operation of Algorithm 2. In this operation, the mapping from a computational mesh to an external component, such as an equation solver, must be updated to reflect the mesh bulk data modifications. One approach is to rebuild the entire mapping from scratch, a potentially costly

```
begin Application Driver
  create MeshMetaData( input EntityRanks )
    populate MeshMetaData from problem specification
    // e.g., parse user input to create parts and fields
  commit MeshMetaData to final state

  create MeshBulkData( input MeshMetaData )
    populate MeshBulkData with initial mesh
    // e.g., read mesh file to create entities and relations
  change MeshBulkData to consistent state

  setup external dependencies on MeshBulkData
    // e.g., equation solvers, output database, code-coupling

  for simulation steps do
    perform computations on MeshBulkData

    revert MeshBulkData to modifiable state
      modify MeshBulkData
      // e.g., load balancing, adaptivity, element death
    change MeshBulkData to consistent state

    update external dependences on MeshBulkData
  end
  destroy MeshBulkData
  destroy MeshMetaData
end
```

Algorithm 2: Example application driver for creating, using, and modifying mesh meta data and mesh bulk data

operation. Another approach is to modify the mapping to reflect the small modification that occurred. To support coordination among components of the application, the mesh bulk data provide information about what modifications were performed while in its most recent modifiable state.

Mesh bulk data modifications are supported through the following process-local atomic operations:

- create mesh entity,
- change mesh entity part memberships,
- destroy mesh entity,
- create mesh entity relation, and
- destroy mesh entity relation.

Other more complex (*i.e.,* higher-level) modification functions are constructed from these atomic operations. Transaction information includes, to some level of detail, what mesh bulk data modifications have occurred.

9.4 Creating, Changing, and Destroying Mesh Entities and Relations

Creating, changing part memberships, and destroying mesh entities and mesh entity relations occurs through process-local atomic operations on the mesh bulk data aggregate. These atomic operations shall have at worst logarithmic time complexity with respect to the number of mesh entities in the local process. The parallel consistency of these operations is resolved when the mesh bulk data is changed from the modifiable state to the consistent state. The time complexity for parallel resolution of these operations should be proportional to the number of created, changed, or destroyed mesh entities and communication overhead.

9.4.1 Mesh entity creation

A mesh $entity_a^J$ is created to be of a mesh entity rank J, to have a unique integer identifier a (unique for the given rank), and to be the member of a specified collection of mesh parts. A mesh entity is created without any relations to other mesh entities; *i.e.*, $Relation\left(entity_a^J\right) = \emptyset$.

Parallel consistency: The created mesh entity is automatically a member of the Owned_P mesh part where P is the local process. When mesh entities are created on different processes with the same mesh entity rank and identifier then these entities are assumed to be duplicates. However, this duplication is not discovered until the parallel consistency of the mesh bulk data is resolved—which occurs when the mesh bulk data is changed to the consistent state. The parallel consistency resolution operation will:

- insert the duplicated mesh entity into the mesh bulk data for mesh entity sharing (Equation 21),
- choose one of the sharing processes to be the unique owner (Equation 19), and
- insert the duplicated mesh entity into the union of mesh parts obtained from all duplicates.

Parallel index: Parallel resolution of mesh entity creation uses a *parallel index* which assigns all potential mesh entity identifiers to one of the parallel processes. During parallel resolution each created mesh entity identifier is communicated to the assigned process. The assigned process then informs the creating processes of any parallel duplication of their created mesh entities. Given this parallel sharing information, the sharing processes choose a process owner and update the shared mesh entity communication information.

9.4.2 Mesh entity destruction

A mesh $entity_a^J$ can only be destroyed if it is not the to-entity of a relation; *i.e.*, $entity \rightarrow entity_a^J = \emptyset$. For example, a node cannot be destroyed on the local process if an element still depends upon it on the local process. As such, if there are such relations, then they must be destroyed before the mesh entity can be destroyed.

Transaction considerations: A request to destroy a mesh entity immediately *deprecates* the mesh entity—it destroys its associated field data and removes the entity from its mesh part memberships and mesh bucket. However, the mesh entity itself is not immediately destroyed. It must persist until the transaction is completed, parallel consistency is resolved, and dependent components are updated. As such, the destroy operation deprecates the mesh entity by destroying its field data and removing it from its mesh part memberships and mesh bucket. Final deletion is postponed until the transaction has been fully processed.

Parallel consistency: The parallel resolution operation removes the destroyed copy of the mesh entity from all communication lists, including lists on other processes. This parallel resolution is only possible if the destroyed mesh entities persists until after parallel resolution has completed. When a shared mesh entity is destroyed, it is assumed that if there are ghost copies of that mesh entity then the intent of those ghost copies has been invalidated and the ghost copies are destroyed. If a shared and owned mesh entity is destroyed on the owning process, but not destroyed on all other sharing processes, then ownership of that mesh entity is transfered to one of the sharing processes.

Identifier recycling: It may be necessary to recycle the integer identifier of a destroyed entity. For example, consider an application that performs local mesh refinement and coarsening to resolve spatially a localized phenomenon which moves through a mesh. After enough refinement and coarsening cycles, the set of possible identifiers will be exhausted if they are not recycled.

9.4.3 Mesh Entity Relation Creation

A mesh entity relation $\left(entity_a^J , entity_b^K , attribute \right)$ is created with a

- from-entity $entity_a^J$,
- to-entity $entity_b^K$ where $J > K$, and
- integer *attribute* value.

The from-entity, to-entity rank K, and attribute value are required to be matched with a single to-entity, as per Equation (5).

Induced part membership: When a mesh entity relation is created, the part memberships of the to-entity are automatically modified to enforce induced part membership specifications (Section 4.3). For example, when an element→node relation is created, the node will automatically become an induced member of the element block mesh part containing the element.

Parallel consistency: If a created mesh entity relation's from-entity is parallel-duplicated, then the closure of the from-entity changes and can become inconsistent among duplicating processes. If a created mesh entity relation's to-entity is parallel-duplicated and changes part memberships due to an induced part membership rule, then the to-entity can become inconsistent among duplicating processes. The parallel consistency resolution operation will add new induced part memberships to parallel-duplicated to-entities as necessary, as well as create new relations and copies of to-entities for parallel-duplicated from-entities as necessary.

Creation of a relation may add the to-entity into a process's $\overline{\text{Owned}_P}$ subset. If the mesh entity already existed in that process's Ghosted_P subset, it must be removed from the Ghosted_P subset to satisfy Equation (22), and introduced into the mesh bulk data's parallel sharing information.

9.4.4 Mesh Entity Relation Destruction

Destruction of mesh entity relations updates induced part memberships as well as parallel consistency, as described below.

Induced part membership: If the to-entity of a destroyed relation has an induced part membership due to that relation, then the to-entity may have to be removed from that mesh part. However, the to-entity may have the same induced part membership from a different mesh entity relation. In this case, the to-entity would not be removed from that mesh part. The induced part memberships of a shared to-entity of a destroyed relation can only be correctly resolved as part of the parallel consistency resolution operation.

Parallel consistency: Destruction of a mesh entity relation can remove the to-entity from the closure of the from-entity. This removal can be recursive; for example, given the simple mesh of $entity_a^J \rightarrow entity_b^K \rightarrow entity_c^L$ then destruction of relation $entity_a^J \rightarrow entity_b^K$ results in the removal of both $entity_b^K$ and $entity_c^L$ from the closure of mesh entity $entity_a^J$. Removal of mesh entities from the mesh entity closures can remove mesh entities from a process's closure, $\overline{\text{Owned}_P}$. Such an occurrence can cause shared mesh entities to be shared no longer and ghosted mesh entities to be needed no longer. Resolution of these complex changes is the responsibility of the parallel consistency resolution operation.

9.4.5 Mesh entity part membership changes:

The part memberships of a mesh entity may be changed by adding that entity to or removing it from membership of selected mesh parts. When a mesh entity is added to a mesh part it is automatically added to all supersets of that mesh part. Likewise, when a mesh entity is removed from a mesh part, it is automatically removed from all subsets of that mesh part. Thus, the added mesh parts, and their supersets, must be disjoint from the removed mesh parts, and their subsets.

Induced memberships: Mesh part membership changes to a mesh $entity_a^I$ will most likely cause induced mesh part membership changes to the to-entities in the $entity_a^I \rightarrow entity$ relations. Such potential changes are resolved by adding new mesh part memberships and removing an existing mesh part membership.

Parallel consistency: Part memberships may be added to or removed from shared mesh entities. These changes may be inconsistent among the sharing processes. Parallel consistency is resolved according to the following two rules.

- Adding a new part membership to a shared mesh entity on any sharing process causes that part membership to be added to all duplicates of that mesh entity.
- Removing an existing part membership from a shared mesh entity is successful if-and-only-if that existing part membership is removed on all sharing processes.

Thus, adding a mesh part membership will always occur but removing a mesh part membership will not occur if any one of the sharing processes does not remove that mesh part membership.

9.5 Transferring Parallel Ownership

Parallel ownership of a mesh entity may be transfered by the owning process to another process. Transfer of ownership does not modify a mesh when viewed as a whole; however, it will modify the parallel distribution of that mesh and memberships in the $\overline{\text{Owned}_P}$ subsets. Ownership transfer is typically performed as part of a parallel dynamic load balancing operation.

Transfer of ownership performs a collective communication to copy the mesh entities, their closures, and associated field data from the previous owner process to the new owner processes. Thus mesh entities are copied or moved among the $\overline{\text{Owned}_P}$ subsets. If a mesh entity is no longer in a $\overline{\text{Owned}_P}$ subset then it is deleted from that process. Resolution of changes to the $\overline{\text{Owned}_P}$ subsets revise the sharing communications lists (Equation 21) accordingly.

Changing the $\overline{\text{Owned}_P}$ membership of a ghosted mesh entity is likely to invalidate the application's intended use for that ghosted mesh entity. As such, the parallel resolution operation destroys all ghosts of mesh entities which have changed $\overline{\text{Owned}_P}$ memberships. An application will have to re-evaluate the ghosting needs for these mesh entities following an onwership-transfer operation.

9.6 Modifying Parallel Ghosting

An application can modify parallel ghosting by adding mesh entities to, or removing mesh entities from, its chosen ghosting subset $\{\text{Ghosted}_P\}_\Phi$ (recall Section 7.2). Such modifications do not change the computational mesh when viewed as a whole. An application provides a collection of add-to and remove-from instructions to a parallel collective operation to allow efficient parallel resolution of the modification operation.

Parallel ghosted mesh entities are automatically removed, during parallel consistency resolution, due to the following changes to the mesh.

- A ghost mesh entity's membership in $\overline{\text{Owned}_P}$ subsets is changed. This can occur due to transfer of parallel ownership (Section 9.5), or due to the destruction of a shared-duplicate of the ghosted mesh entity.
- The relations of a ghosted mesh mesh entity change—potentially invalidating the application's intended use for that ghosted mesh entity. For example, if a mesh entity that was ghosted because it is the neighbor of a shared mesh entity, and a relation in that "neighbor path" is destroyed, then the intended use of that ghosted mesh entity is invalidated.

10 Sierra Toolkit Mesh Implementation

The domain model for a parallel, heterogeneous, and dynamically modifiable unstructured mesh presented in this report is implemented as a component of the Sierra Toolkit. This implementation and its application programmer interface (API) are being developed through an agile software development process. As such this domain model, the derived software design, and API are subject to change as new use-cases are defined and existing use-cases are refined.

The concepts described in the previous sections of this report are considered by the Sierra Toolkit development team to be sufficiently stable to archive in this written report. However, the derived software design and API for these capabilities is subject to change as the collection of use-cases is broadened and performance is optimized. As such documentation for the software design and API currently limited to extensive commentary within the Sierra Toolkit Mesh source code.

The Sierra Toolkit Mesh is available through Sandia National Laboratories' Trilinos repository at http://trilinos.sandia.gov/.

11 Future Development

Continued development of the Sierra Toolkit computational mesh is planned, with many potential ideas for future development. Identification and description of these ideas in this section carries **no** implications as to development priorities, schedule, or even guarantee for follow-through. The concepts for these planned capabilities have not been thoroughly evaluated through multiple use-cases and are not as stable as the domain model presented in the previous sections of this report.

11.1 Thread Parallelism

Two levels of parallel execution are considered: (1) process-parallelism and (2) thread-parallelism within each process. To date, parallel execution efforts have focused exclusively on process-parallelism via a Message Passing Interface (MPI) compliant inter-process communication library. Given new multicore architectures it is necessary to consider thread-parallelism in addition to the existing process-parallelism programming model. The goal for this two-level parallel programming model is to achieve better parallel performance than could be achieved by simply applying the single-level process-parallel programming model to the processing cores of a compute-node. For compute-node architectures with a heterogeneous collection of processing cores, such as those that utilize graphics accelerators, it may not be possible to use a single-level process-parallel programming model. An additional complication that can be present with heterogeneous processing cores is the requirement to explicitly manage the movement of data between distinct portions of the node's or process' memory. For example, data may need to be copied between a node's main memory and a graphics accelerator's internal memory in order for the processing cores of the graphics accelerator to operate on that data. A homogeneous collection of processing cores may appear to have equal access to the process memory. However, such multicore system's cache memory architecture can have significant variations in overall performance with respect to a computation's memory access patterns. As such data layouts and parallel programming models need to be considered when determining how to most effectively use multicore resources.

11.2 Field Data Consistent Units System

Another potential use case is for a field declaration to have mathematical units of measure (*e.g.*, meters, degrees Celsius, kilograms). This potential use case is not currently realized in the domain model or implementation of the SIERRA Toolkit Mesh.

11.3 Entity Ordering

Currently, the entity ordering relation expresses a unique ordering of mesh entities used to ensure mesh results are invariant for meshes with identical data in identical computation environments. This idea can be extended to provide several other services to applications. For instance, a relation can be supplied by the application developer to reduce cancellation in local operations and reorder computations to use cache more efficiently.

11.3.1 Operator Mappings

A simulation generally defines a problem on a particular discretization, uses the discretization to construct a mathematical operator, and then computes an approximation to the problem. As part of this procedure, the application developer constructs an *application mapping*, the process by which a discretization can be used to build an operator A:

$$A : mesh \mapsto operator.$$

By using the SIERRA toolkit, the application developer can translate the mesh into a parallel, consistent data structure that can be used to construct an operator.

The SIERRA Toolkit provides a mapping from mesh entities to owning process and intersection of mesh parts. For instance, given a node in a finite element mesh, the SIERRA Toolkit computes an owning process for the node and maintains a bucket on that process in which the node is stored. Thus, a *Toolkit mapping* can be defined as $S : mesh \rightarrow (process, storage)$. Each entity in the mesh is associated with an owning process and a place in its memory.

The application developer could transform the mesh stored in memory into an operator. For example, the application may require construction of a finite element stiffness matrix. In parallel, the matrix would be distributed among processes in a matrix storage scheme such as compressed sparse row. The source code defines another mapping $T : (process, storage) \mapsto (process, operator\ component)$. Thus, A is the composition of S and T. The operator $(process, operator\ component)$ is then provided to computational kernels to approximate a solution to the problem.

In order to remove mesh evolution-dependent effects from construction of A, a mathematical relation is used to sort mesh entities. The relation well orders $(process, storage)$, implying a bijective ordering map from the integers $Z = [1, ||storage||]$ to the $R : [1, ||storage||] \mapsto storage$. This allows the application developer to use the mesh interface in a straightforward fashion. Rather than creating a fine grain source code mapping, the application developer can use the familiar iterator access pattern: given the nth item in the bucket, construct the mth line in the operator component. Therefore, the application developer creates a map

$$T' : (process, \mathbb{Z}^+) \mapsto (process, operator\ component)$$

. In this way, applications can be described as $A' : T' \circ R^{-1} \circ S$.

11.3.2 Operator Permutations

A more sophisticated mechanism may be implemented that introduces a permutation on $(process, storage)$. For two computational environments i and j, there are two Toolkit mappings S_i and S_j. $S_i = S_j$ if and only if $i = j$. Let $R_{ij} : (process, storage) \mapsto (process, storage)$ be a bijective permutation such that $S_j = R_{ij} \circ S_i$. Then, given $A_i = T \circ S_i$, it is possible to compute $A_j = T \circ R_{ij} \circ S_i$. In other words, given the storage scheme computed by the Toolkit in environment i, R can transform it to the same storage scheme computed in environment j.

An ordering can be computed from the current environment variables (*e.g.* process rank, communicator size) and variables provided to the Toolkit describing the desired environment. Consider a single mesh with $2^{20} = 1,048,576$ nodes, each in the same homogeneous subset and stored in a bucket labeled "nodes." Each node is given an offset into the bucket, a pointer that identifies where in memory the data for the node is stored.

Let two different simulations use this mesh, simulation i on $2^{10} = 1,024$ processes and simulation j on $2^{15} = 32,768$ processes. Assume the mesh is decomposed to be perfectly load balanced in each simulation and each node assigned to a process by integer division. There are two different Toolkit mappings. In this example, let each node be placed on process in a round-robin fashion: *process* = *rank* mod communicator size and let each node be placed in memory in a similar fashion, *pointer* = *rank* div communicator size.

$$\begin{aligned} S_i(rank, id) &= (rank \bmod 1024, (\text{“nodes”}, rank \operatorname{div} 1024)) \\ S_j(rank, id) &= (rank \bmod 32768, (\text{“nodes”}, rank \operatorname{div} 32768)), \end{aligned}$$

where (“nodes”, *rank* div 1024) fully specifies the mesh storage.

From these mappings, a permutation R_{ij} can be constructed so that simulation j will be presented the mesh in exactly the same fashion as simulation i:

$$\begin{aligned} R_{ij}(x,(y,z)) &= S_j \circ S_i^{-1} \\ &= ((1024z + x) \bmod 32768, (y, (1024z + x) \operatorname{div} 32768)). \end{aligned}$$

This mapping is constructed by inverting the current mapping computed by the Toolkit and recomputing the mapping for a new enviornment. This mapping is identical to $R'_{ij} = (rank \bmod 32768, (\text{“nodes”}, rank \operatorname{div} 32768))$. However, this formulation implies the Toolkit can infer $rank = 1024z + x$, which it may not be able to. In the former formulation, the constants are available as parameters that describe the current environment (1024) and the desired environment (32768).

11.4 Mesh Modification Transaction Logging

Mesh modifications potentially affect any capability that maps a mesh to a data structure external to that mesh. For example, after mesh refinement, the mappings between the

mesh and output files, mappings between the mesh and equation solvers, or even mappings between two different meshes may have changed. These mappings can be expensive to compute from scratch but easy to update given a list of incremental changes to the mesh. With sufficient knowledge of the incremental mesh modifications, these mappings could be incrementally updated.

11.4.1 Mesh Modification Transaction

Algorithm 3 depicts an example multi-physics simulation algorithm. In this algorithm, modifications to mesh entities must occur between "begin MeshBulkData modification" and "end MeshBulkData modification." A *mesh modification transaction* is the collective description of these modifications to mesh entities.

```
begin Simulation Algorithm
  while simulating do
    // begin application's transaction
      begin MeshBulkData modification
        Modify MeshBulkData
        // e.g., refine elements, element death, etc.
      end MeshBulkData modification
      // mesh modification is done;
      // but mappings are inconsistent
      Create/update mappings between mesh and external data structures
    // end application's transaction
    Perform simulation
  end
end
```

Algorithm 3: General simulation algorithm

These transactions are serializable with respect to the processes. Each process tracks its own local contribution to the transaction, called a *transaction fragment*. That is, each transaction fragment is assumed to have executed atomically and the order of commitment of these transaction fragments does not affect the resulting mesh. To ensure this property, only the total change of the mesh entity is recorded. Thus, the transaction log is only required to record two states of each modified entity: the beginning state and the final state.

When the transaction is committed at "end MeshBulkData modification", the parallel consistency rules are applied to ensure the final state of each entity is consistent across all processors. By induction, the beginning state of the transaction is always in a consistent state.

A potential use of the transaction log is to determine which changes in the mesh affect mappings to external data structures and update these mappings accordingly. For instance, the element-sides belonging to a part representing a Dirichlet boundary condition may have

changed due to element death. The element death module removed elements, exposing interior faces to the boundary condition. In this case, the application can update its boundary condition mapping by learning from the log which elements were destroyed and which faces were exposed.

11.4.2 Types of Transaction

There are two types of transactions based on how the transaction will be used. The first type, bulk transactions, are used when mapping repairs are too expensive, unstable, or ill-advised for other reasons. Alternatively, incremental transactions are used whenever it is cheaper to repair mappings than to rebuild them.

Bulk Transaction: Individual mesh entities are not logged. Rather, it is assumed that most of the mesh entities owned by a process will be changed. Mesh parts containing changes are noted.

Incremental Transaction: All information from a bulk transaction is included in an incremental transaction. Each mesh entity inserted, deleted, or modified is tracked in the transaction. Through various mechanisms, applications developers can select, iterate, or otherwise access these entities just as any other mesh entity.

11.4.3 Incremental Transactions as State Changes

The incremental transaction records three types of entity changes: insertion, deletion, and modification. Insertion and deletion of entities are straightforward to track. On the other hand, an entity is modified if it matches any one of the following three criteria:

- **Relation is modified.** Any of the relations in the set of relations from the entity is changed, either in to-entity or attribute. For instance, if the local node numbering of a face changes, then the face is modified but the nodes are not.
- **The state of entity changes.** If a mesh entity is added or removed from a mesh part, the mesh entity is considered modified
- **Any entity directly related to the entity is modified.** Continuing the element-side example from above, both elements incident on the face are modified.

The mesh state is independent of the transaction log. The transaction log maintains three sets: Deleted, Inserted, and Modified. The transaction log does not track changes to field data or relations. Mesh bulk data and transaction sets share the same meta data. In this way, the two have the same mesh parts

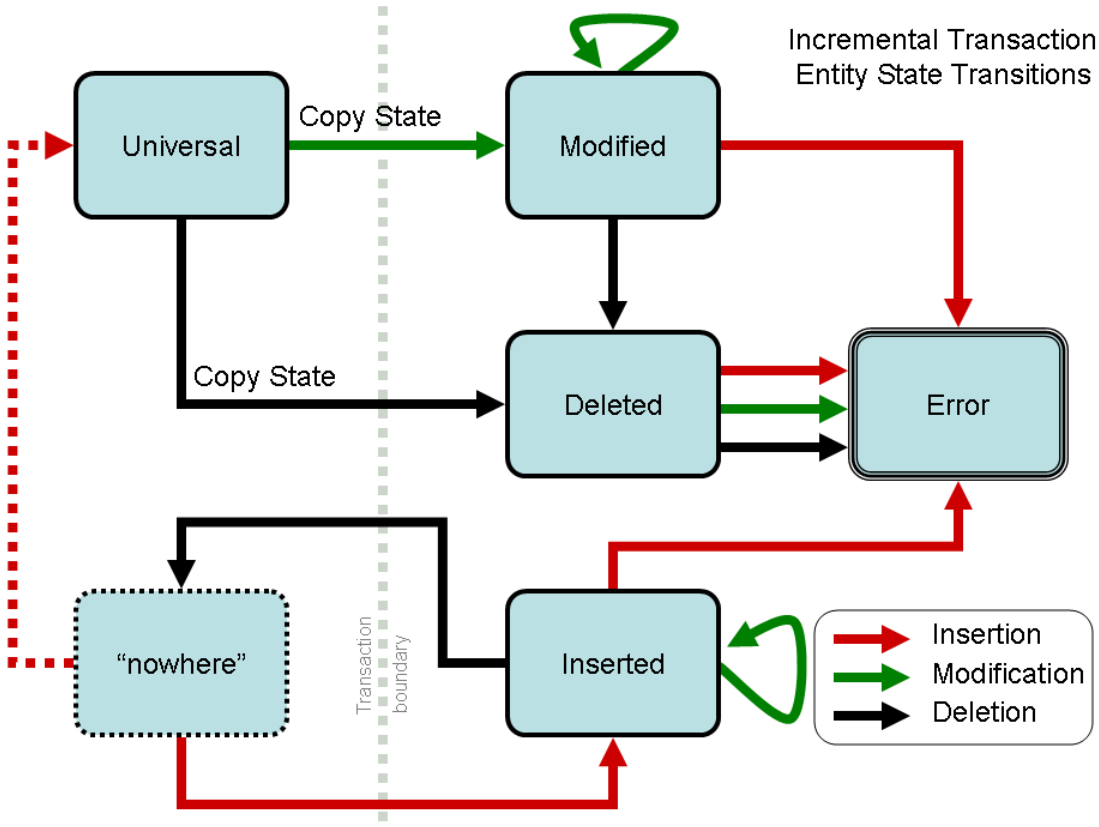


Figure 10. State diagram for serial transactions

When a mesh entity is modified, inserted, or deleted, it may be logged. If the entity is not already in Deleted, Inserted, or Modified, the entity will be placed in the appropriate set along with the parts in the transaction bulk data that describe its current state.

Entity Insert When an entity is inserted into the mesh, it is added to Inserted.

Entity Modification If the entity is a member of Inserted or Modified already, nothing happens. Otherwise, the entity is added to Modified along with the rest of the parts that describe its current state.

Entity Delete If the entity is in Modified, it is removed from this group and placed in Deleted, keeping the bucket parts. If it is in Inserted, the entity is deleted from memory. Otherwise, it is added to Deleted and the rest of the parts that describe its state.

The state diagram in Figure 10 describes how an entity will be stored in the transaction based on how the entity is changed. An entity in the Universal part can be modified or deleted, placing the entity into the transaction log. A created entity is placed in the Universal set (indicated by the dotted line). This entity is also placed in the transaction log.

For instance, let *entity* be in parts A, B, and C but not in any transaction set. If *entity* is added to part D, then *entity* is added to Modified. The state of *entity* at the beginning of the transaction (a member of A, B, and C) is preserved in the transaction log.

11.4.4 Transaction Lifetime

The transaction log can be reset at any time when the mesh is in a parallel consistent state (i.e., no mesh modification is in progress). For bulk transactions, resetting involves erasing the list of stored modified parts. For incremental transactions, resetting the transaction involves two steps. First, the mesh entities in Deleted are finally deleted from mesh bulk data. Next, the remaining parts of the transaction bulk data are emptied without deleting the entities.

Once the change has been committed, the three mesh parts can be used by the application to find which entities were changed and how they were modified. For instance, if a bulk transaction is used, and a mesh part called "Interior Elements" was modified, then it is assumed that any of the entities in the mesh part "Interior Elements" could have been modified. If the transaction is incremental, then individual mesh entities can be iterated over based on querying of mesh parts. In either case, after the end of the bulk data modification, the application can look through the various mesh parts to repair or rebuild mappings as necessary. The data are available until the transaction log is reset.

11.4.5 Parallel Consistency and Ambiguities

If one process modifies a mesh entity, then it will be modified on other processes when the parallel synchronization operation is applied. The transaction log tracks modifications across process boundaries.

Mesh entities can be stored differently in the transaction log depending on the process they are stored and the mechanism by which they were changed. For instance, an entity can be modified on one process and deleted from another simultaneously. On each process, the entity will be handled differently by the transaction log.

References

[1] H. Carter Edwards. Sierra framework version 3: Core services theory and design. Technical report SAND2002-3616, Sandia National Laboratories, Albuquerque, New Mexico 87185, November 2002.

[2] H. Carter Edwards. Managing complexity in massively parallel, adaptive, multiphysics applications. *Engineering with Computers*, 22(3-4):135–156, 2006.

[3] Eric Evans. *Domain-Driven Design, Tackling Complexity in the Heart of Software*. Addison-Wesley Publishing Company, 2004.

[4] Information Technology Industry Council. *Programming Languages — C++, International Standard ISO/IEC 14882*. American National Standards Institute, 11 West 42nd Street, New York, New York 10036, first edition, 1998.

[5] Stefan Lang. Parallel-adaptive simulation with the multigrid-based software framework UG. *Engineering with Computers*, 22(3-4):157–180, 2006.

[6] Stefan Lang and Gabriel Wittum. Large-scale density-driven flow simulation using parallel unstructured grid adaptation and local multigrid methods. *Concurrency and Computation: Practice and Experience*, 2003.

[7] IEEE Std 1003.1, 2004 Edition, <pthread.h>, 2004.

[8] Larry A. Schoof and Victor R. Yarberry. EXODUS II: A Finite Element Data Model. Technical report SAND92-2137, Sandia National Laboratories, Albuquerque, New Mexico 87185, November 1995.

[9] Seegyoung E. Seol and Mark S. Shepard. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Engineering with Computers*, 22(3-4):197–214, 2006.

[10] James R. Stewart and H. Carter Edwards. A framework approach for developing parallel adaptive multiphysics applications. *Finite Elements in Analysis and Design*, 40:1599–1617, 2004.

[11] AOMD home page. http://www.scorec.rpi.edu/AOMD/, July 2005.

[12] Basic linear algebra subprograms (BLAS) website. http://www.netlib.org/blas/, October 2009.

[13] Nvidia cuda home page. http://www.nvidia.com/object/cuda_home.html, October 2009.

[14] Message passing interface (MPI) standard documents. http://www.mpi-forum.org/docs/docs.html, October 2009.

[15] Trellis home page. http://www.scorec.rpi.edu/Trellis/, July 2005.

[16] UG home page. http://cox.iwr.uni-heidelberg.de/~ug/, July 2005.

DISTRIBUTION:

1	MS 0380	David E. Womble, 01540
1	MS 0382	Basil Hassan, 01541
1	MS 0380	Joe Jung, 01542
1	MS 0382	Ted D. Blacker, 01543
1	MS 0828	Anthony A. Giunta, 01544
1	MS 0382	David G. Baur, 01543
1	MS 0382	H. Carter Edwards, 01543
1	MS 0382	Michael W. Glass, 01543
1	MS 0382	Greg D. Sjaardema, 01543
1	MS 0382	Alan B. Williams, 01543
1	MS 0380	James G. Foucar, 01543
1	MS 0382	Kenneth Belcourt, 01543
1	MS 0370	James R. Overfelt, 01433
1	MS 1320	Todd S. Coffey, 01414
1	MS 1313	Russel Hooper, 01414
1	MS 1137	Daniel Sunderland, 06326
1	MS 0380	Kendall H. Pierson, 1542
1	MS 0336	Patrick K. Notz, 1541
1	MS 0382	Kevin D. Copps, 1544
1	MS 0899	Technical Library, 9536

Sandia National Laboratories